

# Annoy Recommendation System

Francesco Bari

Corso di Laurea Magistrale in Computer Science, 2020285  
`francesco.bari.2@studenti.unipd.it`

**Sommario** Il progetto consiste nella realizzazione di un sistema di raccomandazione basato sulla similarità tra oggetti appartenenti ad uno stesso campo di interesse. Per illustrarne i passaggi verrà preso in considerazione il contesto legato al mondo degli indumenti (fashion). Per implementare il sistema verranno quindi utilizzate due Reti Neurali Convolutionali (CNN) e la libreria Annoy rilasciata da Spotify.

**Keywords:** computer vision · recommendation system · CNN

## 1 Introduzione

### 1.1 Scopo del progetto

Lo scopo del progetto risiede nella realizzazione di un sistema di raccomandazione basato sulla similarità tra oggetti appartenenti ad uno stesso campo di interesse. Si è scelto volutamente di rendere il sistema più generico possibile, capace quindi di adattarsi ad un qualsivoglia catalogo relativo per esempio ad un semplice brand piuttosto che ad catalogo di un generico sito di e-commerce. Il caso d'uso preso in esame presenta la realizzazione di un sistema di raccomandazione basato su un dataset contenente capi di abbigliamento di vario tipo, reperibili all'indirizzo:

<https://www.kaggle.com/paramaggarwal/fashion-product-images-dataset>

È stato scelto volutamente l'utilizzo di un catalogo che non presenti etichette (label) riferite ai capi di abbigliamento in esso contenuti (*class-agnostic*). I motivi di questa scelta ricadono nuovamente sulla forte volontà di rendere il sistema più generico possibile, in grado quindi di adattarsi anche a casi d'uso differenti da quello preso in esame.

### 1.2 Struttura del documento

Viene brevemente presentato il contenuto dei capitoli appartenenti al documento.

Il secondo capitolo (§2) descriverà la fase di *Object Detection*, dove verranno reperiti i dati non classificati di un generico catalogo, ai quali verrà applicata

una classificazione (label) ed un ritaglio (crop) relativo unicamente al prodotto stesso, escludendo quindi eventuali volti di modelli e/o rumore in generale.

Il terzo capitolo (§3) descriverà la fase di *Feature Extraction*, dove, a partire dalle immagini ritagliate ed etichettate nelle alla fase precedente, andremo ad estrarre i *feature vectors* riferiti ad ogni prodotto.

Il quarto capitolo (§4) descriverà quindi la fase di *Image Similarity*, dove, a partire dai file contenenti i *feature vectors* estratti da ogni singolo prodotto, andremo a creare un indice di similarità (grazie alla libreria di Spotify - Annoy) e, per un prodotto di *query*, andremo a calcolare i relativi *nearest neighbours*.

Nel quinto capitolo infine (§5) verranno illustrati i risultati di alcune *query*, discussi gli aspetti generali del sistema e verranno presentati alcuni possibili miglioramenti.

### 1.3 Codice di riferimento

Tutto il codice relativo al progetto, oltre ad essere consegnato con questa relazione, potrà essere interamente consultato nel seguente notebook (compilato) e repository (compilabile):

- **Kaggle:** <https://www.kaggle.com/francescobari/annoy-recommendation-system>
- **GitHub:** <https://github.com/cescomuch/annoy-recommendation-system>

## 2 Object Detector

Per cominciare avremo bisogno innanzitutto di reperire un dataset contenente delle immagini relative ad un generico catalogo di un brand piuttosto che ad generico sito di e-commerce.

La maggior parte dei cataloghi relativi a queste realtà sono fruibili in rete, piuttosto che in un sistema proprietario di una azienda, sia in formato JSON che in formato CSV. Il sistema proposto sarà in grado di elaborare indistintamente i due formati (in caso di un file CSV ci ricondurremo al caso di un file in formato JSON).

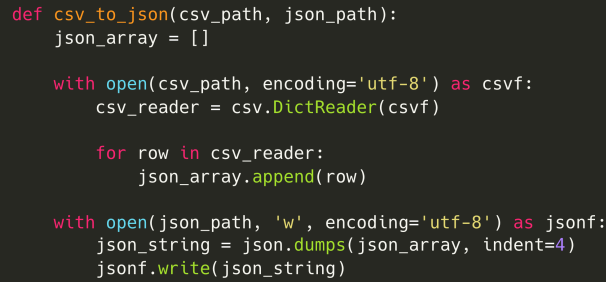
Basterà quindi avere un riferimento alla locazione di tale file per cominciare tutto il processo.

Come detto poc'anzi, il caso d'uso preso in esame presenta la realizzazione di un sistema di raccomandazione basato su un dataset contenente capi di abbigliamento di vario tipo. Tra i vari file disponibili, il dataset preso in considerazione è rappresentato da un file `style.csv`, contenente al suo interno l'informazione relativa alle coppie `id - path` dove:

- `id`: rappresenta un identificativo univoco legato al prodotto;
- `path`: rappresenta l'url relativo alla immagine del prodotto in questione.

A partire da questo file:

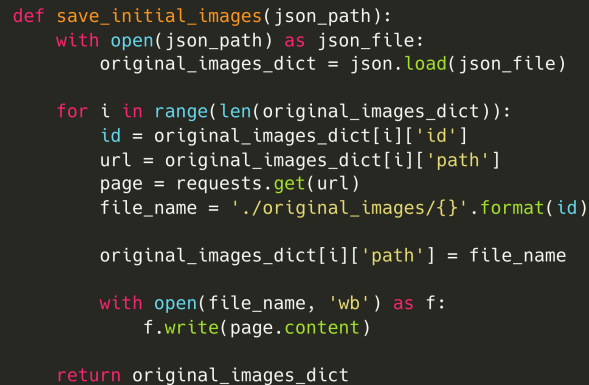
1. eseguiamo la funzione `csv_to_json` per ottenere il catalogo sotto forma di JSON;

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains the Python function `csv_to_json`.

```
def csv_to_json(csv_path, json_path):  
    json_array = []  
  
    with open(csv_path, encoding='utf-8') as csvf:  
        csv_reader = csv.DictReader(csvf)  
  
        for row in csv_reader:  
            json_array.append(row)  
  
    with open(json_path, 'w', encoding='utf-8') as jsonf:  
        json_string = json.dumps(json_array, indent=4)  
        jsonf.write(json_string)
```

**Figura 1.** Metodo `csv_to_json`

2. eseguiamo quindi la funzione `save_initial_images` per reperire tutte le immagini e salvarle in una directory locale.

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains the Python function `save_initial_images`.

```
def save_initial_images(json_path):  
    with open(json_path) as json_file:  
        original_images_dict = json.load(json_file)  
  
    for i in range(len(original_images_dict)):  
        id = original_images_dict[i]['id']  
        url = original_images_dict[i]['path']  
        page = requests.get(url)  
        file_name = './original_images/{}'.format(id)  
  
        original_images_dict[i]['path'] = file_name  
  
        with open(file_name, 'wb') as f:  
            f.write(page.content)  
  
    return original_images_dict
```

**Figura 2.** Metodo `save_initial_images`

Una volta reperite tutte le immagini relative al catalogo, abbiamo bisogno di andare a scegliere il modello da utilizzare nella fase di *Object Detection*. Per fare

questo sfruttiamo un concetto molto potente, ossia il *Transfer Learning* [1]. Il *Transfer Learning* è una metodologia nella quale un *ML Model* sviluppato per una specifica attività viene riutilizzato come punto di partenza per risolverla una seconda attività, normalmente molto simile o comunque correlata alla prima. È una tecnica molto popolare nel mondo del *Deep Learning*, nel quale i *Pre-trained Models* vengono utilizzati come punto di partenza per risolvere problemi di visione artificiale che necessitano di grandi risorse (di tempo e calcolo) attui a sviluppare modelli di reti neurali in grado di risolvere tali problemi.

Nello specifico utilizzeremo **FasterRCNN + InceptionResNet V2**: si tratta di un *Object Detection model* [2] addestrato sul dataset **Open Images V4** con alla base un **Inception Resnet V2** pre-addestrato sul dataset **ImageNet** come *Image Feature Extractor*.

**FasterRCNN** è un modello pubblicato da Google (proprietaria di Tensorflow) che si avvale appunto dell'omonima architettura **FasterRCNN**. Il modulo utilizza come algoritmo per scegliere i *detection boxes* da mantenere rispetto a quelli da scartare la *non-maxima suppression*.

Dopo questa soppressione, la rete sarà in grado di rilevare fino a 100 oggetti all'interno di una immagine, appartenenti ad un totale di 600 categorie.

La lista completa delle categorie è disponibile al seguente dendrogramma:

[https://storage.googleapis.com/openimages/2018.04/bbox\\_labels\\_600\\_hierarchy\\_visualizer/circle.html](https://storage.googleapis.com/openimages/2018.04/bbox_labels_600_hierarchy_visualizer/circle.html)

Di queste 600 categorie noi andremo nello specifico a filtrare solo quelle di nostro interesse (categoria fashion), ma è chiaro come il sistema possa adattarsi molto bene a tutti gli altri domini applicativi.

Basterà quindi cambiare la lista delle classi di nostro interesse per aggiungere/rimuovere tali classi dal nostro sistema di filtraggio.

```
class_list = ["Clothing", "Cowboy hat", "Sombrero", "Sun hat", "Scarf",
             "Skirt", "Miniskirt", "Jacket", "Fashion accessory", "Glove",
             "Baseball glove", "Belt", "Sunglasses", "Tiara", "Necklace",
             "Sock", "Earrings", "Tie", "Goggles", "Hat", "Fedora", "Handbag",
             "Watch", "Umbrella", "Glasses", "Crown", "Swim cap", "Trousers",
             "Jeans", "Dress", "Swimwear", "Brassiere", "Shirt", "Coat", "Suit",
             "Footwear", "Roller skates", "Boot", "High heels", "Sandal",
             "Sports uniform", "Luggage & bags", "Backpack", "Suitcase",
             "Briefcase", "Helmet", "Bicycle helmet", "Football helmet"]
```

**Figura 3.** Lista delle classi di nostro interesse

L'input del modulo è una immagine a tre canali di grandezza variabile (questa dovrà essere trasformata in un tensore e normalizzata nell'intervallo  $[0,1]$ ).

Come output invece, avremo tre **Python dictionaries**:

- **detection\_boxes**: contenente le coordinate (nell'ordine ymin, xmin, ymax, xmax) di tutti *detection boxes* rilevati nell'immagine di partenza;
- **detection\_class\_names**: contenente i nomi delle varie classi corrispondenti ad ogni *detection boxes* rilevato nell'immagine di partenza;
- **detection\_scores**: contenente le percentuali di *accuracy* delle varie previsioni effettuate, corrispondenti ad ogni *detection boxes* rilevato nell'immagine di partenza.

Quindi, riassumendo, per utilizzare l'architettura appena descritta dovremo:

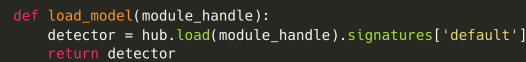
1. Richiamare il modello:



```
module_handle_detector = "https://tfhub.dev/google/faster_rcnn/openimages_v4/inception_resnet_v2/1"
detector = hub.load(module_handle).signatures['default']
```

**Figura 4.** Chiamata a Tensorflow Hub per utilizzare l'architettura **FasterRCNN**

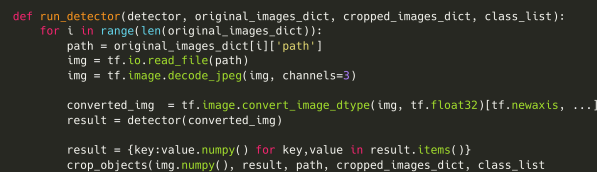
2. Caricarlo attraverso il metodo `load_model`:



```
def load_model(module_handle):
    detector = hub.load(module_handle).signatures['default']
    return detector
```

**Figura 5.** Caricamento del modello

3. Chiamare il metodo `run_detector` su ogni immagine che vogliamo processare:



```
def run_detector(detector, original_images_dict, cropped_images_dict, class_list):
    for i in range(len(original_images_dict)):
        path = original_images_dict[i]['path']
        img = tf.io.read_file(path)
        img = tf.image.decode_jpeg(img, channels=3)

        converted_img = tf.image.convert_image_dtype(img, tf.float32)[tf.newaxis, ...]
        result = detector(converted_img)

        result = {key:value.numpy() for key,value in result.items()}
        crop_objects(img.numpy(), result, path, cropped_images_dict, class_list)
```

**Figura 6.** Metodo `run_detector`

- Esso trasforma l'immagine in un tensore dopo averla normalizzata;
  - Chiama il detector (**FasterRCNN**) sull'immagine;
  - Salva i risultati dentro **result**;
  - Chiama il metodo **crop\_objects**.
4. Nel metodo **crop\_objects** quindi, vengono presi i risultati della detection (i *detection\_boxes*) e, se hanno una percentuale di confidenza di rilevazione superiore ad un certo valore (per noi del 60%) ed appartengono ad una classe del contesto fashion, vengono ritagliati e salvati con l'informazione legata alla loro classe di appartenenza:

```
def crop_objects(img, result, path, cropped_images_dict, class_list, max_boxes=3, min_score=0.6):
    image = Image.fromarray(img)
    width, height = image.size
    for i in range(min(len(result['detection_boxes']), max_boxes)):
        if (result['detection_scores'][i] >= min_score):
            detected_class = "{}".format(result["detection_class_entities"][i].decode("ascii"))
            if (detected_class in class_list):
                ymin = int(result['detection_boxes'][i][0]*height)
                xmin = int(result['detection_boxes'][i][1]*width)
                ymax = int(result['detection_boxes'][i][2]*height)
                xmax = int(result['detection_boxes'][i][3]*width)
                crop_img = img[ymin:ymax, xmin:xmax].copy()
                crop_img = cv2.cvtColor(crop_img, cv2.COLOR_RGB2BGR)
                detected_class = detected_class.replace(" ", "-")
                outfile_name = os.path.basename(path).split('.')[0] + "-" + str(i) + "_" + detected_class
                cv2.imwrite("./cropped_and_labeled_images/{}.jpg".format(outfile_name), crop_img)
                cropped_images_dict[outfile_name] = detected_class
```

Figura 7. Metodo **crop-object**

### 3 Feature Extractor

Come nel caso della *Object Detection*, anche per la *Feature Extraction* viene utilizzato un modello pre-addestrato.

In questa istanza viene utilizzato l'estrattore **MobileNet V2** [3]. Esso è sempre addestrato sul dataset **ImageNet**.

**MobileNet V2** è una famiglia di reti neurali, progettata per essere efficiente in qualsiasi contesto (soprattutto mobile) ed in ogni attività, che sia di classificazione o comunque collegata ad essa. L'implementazione da noi utilizzata della **MobileNet V2** utilizza un *depth multiplier* pari ad 1.0 e prende in input immagini di dimensione 224x224 pixels.

L'output del modello sarà un batch di *feature vectors*.

Per ogni immagine di input, il *feature vector* sarà composto da 1280 *features*. Lo scopo del nostro script sarà quindi quello di estrarre le *features* a partire dalle immagini ottenute nella fase precedente di *Object Detection*.

Nel dettaglio lo script:

1. Richiama il modello:

```
module_handle_extractor = "https://tfhub.dev/google/imagenet/mobilenet_v2_140_224/feature_vector/4"
detector = hub.load(module_handle).signatures['default']
```

**Figura 8.** Chiamata a Tensorflow Hub per utilizzare l'architettura MobileNet V2

2. Chiama il metodo `get_feature_vectors`:

```
def get_feature_vectors(module_handle):
    module = hub.load(module_handle)

    for filename in glob.glob('./cropped_and_labeled_images/*.jpg'):
        img = load_img(filename)
        features = module(img)
        feature_set = np.squeeze(features)
        outfile_name = os.path.basename(filename).split('.')[0] + ".npz"
        out_path = os.path.join('./feature_vectors', outfile_name)
        np.savetxt(out_path, feature_set, delimiter=',')
```

**Figura 9.** Metodo `get_feature_vectors`

- Esso in prima istanza reperisce l'immagine derivante dalla operazione di *Object Detection* (metodo `load_img`):

```
def load_img(path):
    img = tf.io.read_file(path)
    img = tf.io.decode_jpeg(img, channels=3)
    img = tf.image.resize_with_pad(img, 224, 224)
    img = tf.image.convert_image_dtype(img, tf.float32)[tf.newaxis, ...]

    return img
```

**Figura 10.** Metodo `load_img`

- Successivamente chiama l'*extractor* per calcolare ed estrarre effettivamente gli *image feature vectors* delle immagini;
- Effettua poi una operazione di *squeeze*, che elimina tutte le *single-dimensional-entries*, fornendo quindi un array unidimensionale (un vettore);
- Salva i file `.npz` contenenti gli *image feature vectors*.

## 4 Image Similarity

### 4.1 Annoy

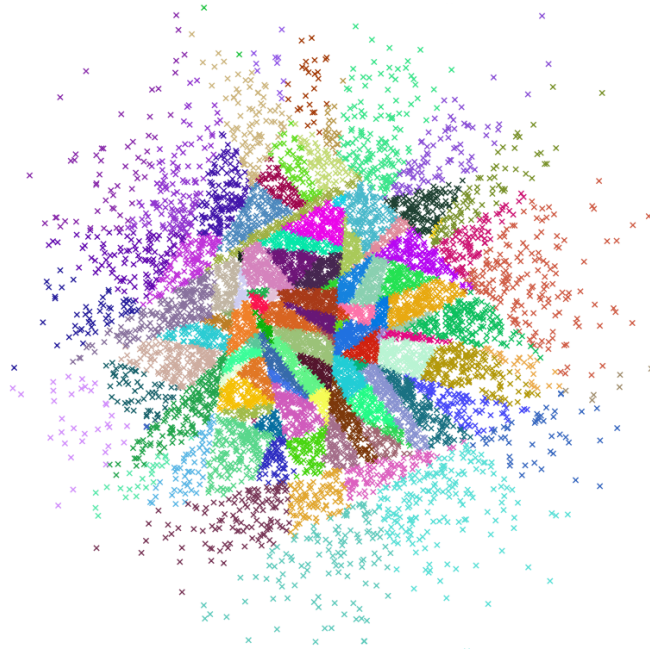
Innanzitutto presento brevemente la libreria principale alla base della *Image Similarity*.

**Approximate Nearest Neighbors Oh Yeah** [4] è una libreria di Spotify, scritta in C++ (compatibile con Python) che, dato un punto di *query* all'interno di un certo insieme (*Annoy Index*), ne cerca i corrispettivi punti più vicini.

Oltre ad essere una tra le librerie riguardanti il problema di approssimazione dei *nearest neighbors* più veloce in tempi di esecuzione e ricerca [5], ha anche la capacità di utilizzare i file statici come indici. Questo significa che può condividere l'indice tra i vari processi in esecuzione. Qualsiasi processo sarà quindi in grado di caricare l'indice in memoria e sarà in grado di eseguire le ricerche immediatamente.

Inoltre è in grado di disaccoppiare la creazione degli indici dal loro caricamento, in modo tale da poter passare gli indici come file, così da poterli mappare rapidamente in memoria.

Anche per quanto riguarda l'utilizzo di quest'ultima, **Annoy** promette buone prestazioni, e quindi pochi sprechi.



**Figura 11.** Rappresentazione visuale dell'indice di Annoy



## 4.2 Similarity

Per quanto riguarda invece lo script designato al calcolo dei *nearest neighbours*, esso prenderà in input i *feature vectors* risultanti dalla *Feature Extraction*, costruirà l'indice di Annoy relativo a tutti i *feature vectors* ricevuti, e, dato un prodotto di *query*, ritornerà i prodotti con score di rilevanza più alto rispetto alla stessa *query*.

Essi saranno dunque ulteriormente filtrati in base alla classificazione fatta con la *Object Detection* per assicurare risultati ancora più rilevanti (ossia ottenere solo i *nearest neighbours* con score di rilevanza maggiore appartenenti alla stessa classe riferita all'immagine di *query*).

Tutto questo può essere riassunto con i seguenti passaggi, ognuno corredato dal relativo *snippet* di codice Python:

1. Inizialmente, vengono presi in input tutti i *feature vectors* risultanti dalla *Feature Extraction* e vengono inseriti nell'indice di Annoy (attraverso il metodo `add_item`) predisposto per l'*angular distance and similarity*;

```
def get_annoy_index(dims, file_index_to_file_vector, file_index_to_product_id):
    allfiles = glob.glob('./feature_vectors/*.npz')
    t = AnnoyIndex(dims, metric='angular')
    for i, file in enumerate(allfiles):
        file_vector = np.loadtxt(file)
        file_name = os.path.basename(file).split('.')[0]
        file_index_to_file_vector[i] = file_vector
        file_index_to_product_id[i] = file_name
        t.add_item(i, file_vector)
    return t
```

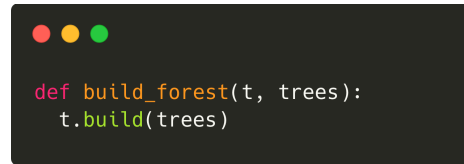
Figura 12. Metodo `get_annoy_index`

2. Successivamente, viene aggiunta la *query* specifica della quale vogliamo calcolare gli oggetti simili, ed il relativo *feature vector* all'indice (sempre attraverso il metodo `add_item`);

```
def add_items(t, file, file_index_to_file_vector, file_index_to_product_id):
    position = len(file_index_to_product_id)
    file_vector = np.loadtxt(file)
    file_name = os.path.basename(file).split('.')[0]
    file_index_to_file_vector[position] = file_vector
    file_index_to_product_id[position] = file_name
    t.add_item(position, file_vector)
```

Figura 13. Metodo `add_items`

- Viene costruito (*build*) l'indice di Annoy vero e proprio, formato da 10.000 alberi;



```
def build_forest(t, trees):
    t.build(trees)
```

Figura 14. Metodo build\_forest

- Infine, vengono calcolati i 5 migliori *nearest neighbours* tramite la formula della *cosine similarity* e viene salvato il tutto in un file JSON contenente i risultati.



```
def score_calculation(n_nearest_neighbors, file_index_to_product_id, file_index_to_file_vector, t, nearest_id, json_output_path):
    named_nearest_neighbors = {}
    final_dict = {}

    last_index = list(file_index_to_product_id.keys())[-1]
    nearest_neighbors = t.get_nns_by_item(last_index, n_nearest_neighbors)

    for i, j in enumerate(nearest_neighbors):
        similarity = 1 - spatial.distance.cosine(file_index_to_file_vector[last_index], file_index_to_file_vector[j])
        rounded_similarity = int((similarity * 10000)) / 10000.0
        nearest_id[file_index_to_product_id[j]] = rounded_similarity

    for id, score in nearest_id.items():
        query_category = file_index_to_product_id[last_index].split("_")
        similar_category = id.split("_")

        if (similar_category[1] == query_category[1]):
            final_dict[id] = score

    final_dict.pop(file_index_to_product_id[last_index])
    named_nearest_neighbors.append({
        'original_product_id': file_index_to_product_id[last_index],
        'similar_products_id': final_dict})

    with open(json_output_path, 'w') as out:
        json.dump(named_nearest_neighbors, out)
```

Figura 15. Metodo score\_calculation

All'interno dello script sono presenti due parametri principali:

- **n\_trees**: indica il numero di alberi (nel nostro caso 10.000). Viene fornito durante la fase di compilazione e influisce sul tempo di compilazione stesso e sulla dimensione dell'indice. Un valore maggiore darà risultati più accurati, ma indici di più grandi dimensioni.
- **search\_k**: indica il numero di nodi da ispezionare durante la ricerca. Esso è pari a il numero di alberi moltiplicato per il numero di *nearest neighbours* cercati (nel nostro caso  $10.000 \cdot 5 = 50.000$ ). Viene fornito a *runtime* ed influisce sulle prestazioni di ricerca. Un valore più grande darà risultati più accurati, ma richiederà più tempo per ritornare i valori cercati.

Quello che Spotify consiglia è di impostare `n_trees` il più grande possibile (ovviamente in base alla quantità di memoria disponibile) e di impostare manualmente `search_k` (quindi non più come `n_trees · search_k`, ma come costante) il più grande possibile, rispettando quelli che possono essere i vincoli di tempo riferiti alla *query* effettuata [6].

La libreria di Annoy utilizza proiezioni casuali [7] per costruire un albero. Ad ogni nodo intermedio di quell'albero viene scelto un iperpiano casuale che divide lo spazio in due sottospazi. Questo iperpiano viene scelto campionando due punti dal sottoinsieme e prendendo l'iperpiano equidistante da essi. Questo processo viene ripetuto `k` volte, in modo tale da ottenere una *forest*, ossia un insieme di alberi. Viene poi calcolata la distanza (*cosine distance*) impacchettando i dati in interi a 64-bit ed utilizzando primitive di conteggio di bit già integrate nella libreria per rendere il tutto il più veloce possibile.

## 5 Conclusioni

Vengono allegate nella cartella "Appendice A" delle immagini contenenti alcuni risultati in merito a varie *query* sottoposte al sistema.

I primi due esempi (*watches*) mostrano come cambiano i relativi risultati per oggetti appartenenti ad una stessa classe.

Una prima importante considerazione riguarda le variabili d'ambiente: si è infatti cercato di rendere il sistema più personalizzabile possibile, andando a parametrizzare il più possibile gli argomenti relativi ai vari script.

Dunque, cambiando i seguenti parametri, si potrà facilmente adattare il sistema alle proprie esigenze. In particolare:

- Per quanto riguarda la *Object Detection*:
  - Il percorso del file CSV (`csv_path`);
  - Il percorso del file JSON (`json_path`);
  - Il modello scelto per questa fase (`module_handle`);
  - La lista delle classi di nostro interesse (`class_list`).
- Per quanto riguarda la *Feature Extraction*:
  - Il modello scelto per questa fase (`module_handle`).
- Per quanto riguarda la *Image Similarity*:
  - La *query* scelta per l'interrogazione del modello (`query_path`);
  - Il percorso del file JSON di output (`json_output_path`);
  - Il numero di *nearest neighbours* che vogliamo ricercare (`n_nearest_neighbours`);
  - Il numero di alberi da utilizzare per la ricerca (`n_trees`);
  - Il numero di nodi da ispezionare durante la ricerca (`search_k`).

Una ulteriore considerazione, come accennato nel §1, riguarda la volontà di rendere il sistema più generico possibile (*class-agnostic*), capace quindi di adattarsi ad un qualsiasi catalogo relativo ad una compagnia, un brand piuttosto che ad un

sito di e-commerce. Basterà infatti avere un dataset contenenti immagini correlate di un identificativo univoco per sfruttare appieno le potenzialità del sistema.

Per quanto riguarda le prestazioni del sistema, i tempi di esecuzione cambiano radicalmente in base alla scelta di eseguire gli script su CPU rispetto ad una GPU. I tempi (riferiti all'esecuzione in ambiente Cloud con l'utilizzo di una GPU dalle medie prestazioni) mostrano quanto segue:

1. **Reperimento immagini:** vincolato dalla propria connessione internet (nell'ordine di pochi minuti con una semplice connessione a 20 MB/s in download);
2. **Reperimento dei modelli pre-addestrati:** anch'esso vincolato dalla propria connessione (resta nell'ordine di pochi secondi con connessioni a 20 MB/s in download);
3. **Object Detection:** circa 1 secondo per ogni immagine processata (in questo senso è la parte più onerosa in assoluto dell'intero sistema. La buona notizia è che essendo una operazione che va svolta "una tantum" per l'intero catalogo, non sarà più necessario eseguirla una seconda volta. Se si volessero aggiungere nuovi prodotti basterà compiere l'*Object Detection* su quest'ultimi);
4. **Feature Extraction:** nell'ordine dei millisecondi;
5. **Creazione indice di Annoy:** nell'ordine dei millisecondi;
6. **Interrogazione del sistema:** circa 1 secondo per ogni interrogazione (*query*).

L'esecuzione in ambiente non Cloud (ossia su CPU), non solo è sconsigliata, ma potrebbe addirittura impiegare diverse ore per terminare.

Per quanto riguarda i requisiti in spazio di archiviazione:

1. **Reperimento immagini:** circa 200 MB di immagini originali;
2. **Object Detection:** circa 250 MB di immagini ritagliate;
3. **Feature Extraction:** circa 22 MB;
4. **Indice di Annoy:** pochi KB.

Per quanto riguarda infine i possibili miglioramenti potrebbero essere considerati i seguenti topic:

- Addestramento degli ultimi strati dei modelli utilizzati (*fine-tuning*) con esempi più rilevanti legati al proprio dominio applicativo;
- Inserimento del sistema in ottiche legate al *Continual Learning* e all'*Online-Learning*;
- Studio di possibili modifiche del sistema legate alla costruzione di *multi-input* e *mixed-data Models*;
- Inserimento dell'architettura in un reale sistema aziendale/professionale (*deploy*).

## Riferimenti bibliografici

1. Transfer Learning, <https://cs231n.github.io/transfer-learning/>
2. Faster R-CNN, [https://tfhub.dev/google/faster\\_rcnn/openimages\\_v4/inception\\_resnet\\_v2/1](https://tfhub.dev/google/faster_rcnn/openimages_v4/inception_resnet_v2/1)
3. Mobile Net, [https://tfhub.dev/google/imagenet/mobilenet\\_v2\\_140\\_224/feature\\_vector/4](https://tfhub.dev/google/imagenet/mobilenet_v2_140_224/feature_vector/4)
4. Spotify - Annoy, <https://github.com/spotify/annoy>
5. ANN Benchmarks, <https://github.com/erikbern/ann-benchmarks>
6. Annoy Tradeoffs, <https://github.com/spotify/annoy#tradeoffs>
7. Random projection, [https://en.wikipedia.org/wiki/Locality-sensitive\\_hashing#Random\\_projection](https://en.wikipedia.org/wiki/Locality-sensitive_hashing#Random_projection)